Repast HPC Manual

Nick Collier

November 23, 2010

1 Introduction

Repast for High Performance Computing (Repast HPC) is an agent-based modeling and simulation (ABMS) toolkit for high performance distributed computing platforms. Repast HPC is based on the principles and concepts developed in the Repast Simphony toolkit. Repast HPC is written in C++ using MPI for parallel operations, and makes extensive use of the boost (http://boost.org) library.

Repast HPC is intended for users with:

- Basic C++ expertise
- Access to high performance computers
- A simulation amenable to a parallel computation. Simulations that consist of many local interactions are typically good candidates.

Models can be written either in C++ or by using a set of "Logo-style" C++ objects that are provided in Repast HPC's "ReLogo" modules.

2 Core Concepts

2.1 Repast Simphony

Repast HPC implements the core Repast Simphony concepts and features.

Agents are implemented as Objects; more specifically, they are created as C++ classes. An agent's state is represented by the field variables of those classes and agent behavior by methods in those classes. For example, a bird agent in a flock simulation might be implemented as a C++ Bird class with field variables for its heading and speed. A fly method might move the bird some distance according the speed and heading.

```
1  class Bird {
2
3  private:
4     float heading;
5     float speed;
6
7  public:
8     ...
9     void fly();
10 }
```

Listing 1: An Example Agent

The simulation proceeds using a schedule. At each iteration of the simulation, the next event is popped of the schedule queue and executed. Repast HPC implements a dynamic discrete-event scheduler with conservative synchronization. The user schedules events to occur at a specific *tick* and the ticks determine the relative order in which the events occur. For example, if event A is scheduled to occur at tick 3, and event B is scheduled for tick 5, then A will execute before B.

A *Context* is used to encapsulate the population of agents. The Context implements the semantics of a logical set; each context may contain only a single instance of each agent. As agents are created they are added to a Context and when they die, they are removed from that Context. A Context may have one or more *Projections* associated with it. A Projection imposes a relational structure on the agents in the Context. For example, a grid projection puts agents into a grid matrix where each agent occupies some cell location in the grid; alternatively, a network projection allows agents to be linked in an arbitrary network of relationships with each other. An agent in a context automatically participates in the projections associated with that Context. When an agent is added to the Context, it becomes a member of the projections associated with the context. For example, if a Context has a network projection then any agents added to the Context become vertices in the network projection. Repast HPC implements 3 types of projections: A grid, a continuous

space and a network.

A Repast HPC simulation is thus composed of agents, one or more contexts containing these agents, and zero or more projections. The user is responsible for writing the agent code; Repast HPC provides the context and projection implementations. Simulation execution typically consists of getting the next event that the user has scheduled and executing that event. The event will invoke agent behavior that uses one or more of the projections. For example, at every iteration of a simulation, each agent might make network links, or poll the agents in neighboring grid cells.

2.2 Parallel Simulation

2.2.1 Parallel Agents

RepastHPC is designed for a parallel environment in which many processes are running in parallel and memory is *not* shared across processes. The agents themselves are distributed across processes. Each process is responsible for the agents *local* to that process. An agent is local to process when the process executes the code that represents the local agent's behavior. Copies of *non-local* agents may reside on a process, allowing agents to be shared across processes. Local agents can then interact with these copies.

As a simple example, suppose a user creates a simulation distributed across 2 process, P1 and P2. Each process creates 10 agents and has its own schedule, which executes events. These events ultimately resolve to method calls on the 10 agents on that process. The 10 agents on P1 are local to P1. Only code executing on P1 changes these agents' states. Similarly, the 10 agents on P2 are local to P2, and only code executing on P2 changes their states. P1 then requests a copy of agent A2 from P2. The Context on P1 now contains the copy of A2. A2 is non-local with respect to P1. The code that runs on P1 should not change the state of A2. Agents on P1 can query the state of A2 when executing their behavior, but the copy of A2 remains unchanged. Of course, the original A2 may change on P2 and in that case, Repast HPC can synchronize the state change between processes.

In order for agents to be distinguished from each other, both in a context and across processes, each agent must have a unique id. This id is represented in the AgentId class.

```
1 class AgentId {
```

```
2     ....
3
4  public:
5     /**
6      * Creates an AgentId.The combination of the
7      * three parameters should uniquely identify the agent.
8      *
9      * @param id the agent's id
10     * @param startProc the rank of the agent's starting
          process
11     * @param agentType the agent's type (user defined)
12     */
13    AgentId(int id, int startProc, int agentType);
14    virtual ~AgentId();
15
16    /**
17     * Gets the id component of this AgentId.
18     *
19     * @return the id component of this AgentId.
20     */
21    int id() const;
22
23    /**
24     * Gets the starting rank component of this AgentId.
25     *
26     * @return the starting rank component of this AgentId.
27     */
28    int startingRank() const;
29
30    /**
31     * Gets the agent type component of this AgentId.
32     *
33     * @return the agent type component of this AgentId.
34     */
35    int agentType() const;
36    /**
37     * Gets the current process rank of this AgentId. The
          current rank
38     * identifies which process the agent with this AgentId is
39     * currently on.
40     *
```

```
41      * @return the current process rank of this AgentId.
42      */
43     int currentRank() const;
44     /**
45      * Sets the current process rank of this AgentId. The
           current rank
46      * identifies which process the agent with this AgentId is
47      * currently on.
48      *
49      * @param val the current process rank
50      */
51     void currentRank(int val);
52     /**
53      * Gets the hashcode for this AgentId.
54      *
55      * @return the hashcode for this AgentId.
56      */
57     std::size_t hashcode() const;
58 }
```

Listing 2: AgentId Class

An AgentId has 4 components:

- id - an user specified numeric id. All agents created on the same process should have a different id.
- agent type - an int signifying the type of the agent. An agent's type corresponds to the agent's C++ class. For example, Bird may have type 0 and Wolf type 1.
- starting rank - the rank of the process on which the agent was created.
- current rank - the rank of the process that the agent is local to. When an agent its created is starting rank and current rank will be identical. However, if the agent moves- that is, if it is transferred, *not copied*- between processes then the current rank will be updated to match the new process.

The AgentId also returns a hashcode() that can be used in unordered stl maps. The structs HashId and AgentHashId found in AgentId.h can be used to parameterize such maps or sets for use with agents and AgentIds.

All RepastHPC agents are required to implement the Agent interface, which simply

provides the AgentId for that agent.

```
class Agent {
public:
   virtual ~Agent() {}
   virtual AgentId& getId() = 0;
   virtual const AgentId& getId() const = 0;
};
```

Listing 3: Agent Interface

2.2.2 Cross-process Communication and Synchronization

Because Repast HPC simulations are distributed across multiple processes, cross-process communication and synchronization of the simulation state across those processes is often necessary. In particular cross-process communication and synchronization are necessary:

- when one process requires copies of agents from another process. This is often necessary in order to "stitch" the global pan-process model into a coherent whole.
- when a process contains non-local agents (or edges) that have been copied from another process, and the copies must be updated with the latest state from the original.
- when grid and space buffers need to be updated. (See below for the details of grid and space buffers.)
- when an agent must be moved completely from one process to another. This can occur as the result of grid movement as explained later below.

Repast HPC automatically addresses most of these conditions. To accommodate the means used by Repast HPC to achieve cross-process communication, the programmer must provide the serialization type code needed to extract the agent state and package it for transfer, and then unpack the transferred package and create or update the appropriate agent from it. This packing and unpacking is known in RepastHPC as the *Package* pattern. It occurs for all sorts of synchronization although the details may differ depending on the type of synchronization (e.g. copying agents the first time vs. updating existing copied agents).

The Package pattern consists of a Package that contains the state to communicate to another processes, a Provider that provides a Package given an agent or edge, and a Receiver that receives the Package, unpacks it and creates or updates an agent or edge from the Package. Here is an example of the Package pattern; in this case, an agent is being copied from one process to another, so the Receiver creates a new agent rather than updating an existing one:

```
class Human {
private: bool _infected;
AgentId id;
public: Human(AgentId id, bool infected) :
    _infected(infected), _id(id){}
};
```

Listing 4: The Agent

The agent here is a Human from the example Zombies model. A human's state is composed of its id and whether or not it is infected. Taken together these two fully describe a Human. The package then needs to encapsulate this state.

```
struct HumanPackage {
template<class Archive>
void serialize(Archive& ar, const unsigned int version) {
   ar & id;
   ar & rank;
   ar & type;
   ar & infected;
}

int id, rank, type; bool infected;
repast::AgentId getId() const {
   return repast::AgentId(id, proc, type);
};
```

Listing 5: The Package

The HumanPackage contains the unique components of a Human's AgentId and the bool infected. In order to identify the agent that the package is for, all packages are required to implement:

```
repast::AgentId getId() const;
```

And so the Human package does so here. Lastly the HumanPackage implements:

```
template<class Archive>
void serialize(Archive& ar, const unsigned int version)
```

This method performs that actual serialization to an Archive. It is this Archive that is then passed via MPI to the other process. Repast HPC uses boost MPI for almost all of its MPI related functionality; more on it can be found at `http://www.boost.org/doc/libs/1_44_0/doc/html/mpi.html`, and more on boost serialization at `http://www.boost.org/doc/libs/1_44_0/libs/serialization/doc/index.html`. However, the general idea is quite simple: any types that are serializable by boost can be serializes using "ar & " followed by the variable. Types can be made serializable using the serialize method above.

```
1 void Provider::providePackage(Human* agent,
2     std::vector<HumanPackage>& out) {
3
4    AgentId id = agent->getId();
5    HumanPackage package = {id.id(), id.startingRank(),
6       id.agentType(),agent->infected()};
7    out.push_back(package);
8 }
```

Listing 6: The Provider

The Provider takes a Human and creates a package from its state. Providers will typically add the created package to a vector as is done here.

```
1 Human* Receiver::createAgent(HumanPackage& package) {
2    return new Human(package.getId(), package.infected));
3 }
```

Listing 7: The Receiver

The Receiver takes the package and makes a Human agent out of it.

This Package pattern applies both to simulations written in straight C++ and those written in the logo-like C++. The details and signatures of the provider and receiver will vary slightly, but the idea remains the same.

3 Repast HPC Simulation Components

This section describes the simulation components implemented by Repast HPC, focusing in particular on how they work in the parallel distributed environment. Additional details about the components can be found in the API documentation. These components are used both when writing a simulation using C++ and to a much lesser extent when writing logo-like C++ simulations, especially for the parallel projections.

3.1 Repast Process

The Repast Process component together with its associated functions manages interprocess communcation and non-local (copied) agent synchronization. The RepastProcess is implemented as a singleton (one per process). It can be retrieved with

```
RepastProcess* rp = RepastProcess::instance();
```

It has methods for getting the total number of processes in the simulation and getting the current process. Associated functions allow the user to request agents from other processes. These agents then become non-local copied agents in the calling process.

```
template<typename T, typename Package, typename Provider,
   typename Receiver>
void requestAgents(SharedContext<T>& context, AgentRequest&
   request,  Provider& provider, Receiver& creator)
```

This uses the Package pattern to package, provide and receive the agents. The AgentRequest encapsulates the request and the context is the simulation context (as described above). An example,

```
1 AgentRequest request(RepastProcess::instance()->rank());
2 // get 10 random agents from other process
3 for (int i = 0; i < 10; i++) {
4    AgentId id = createRandomOtherRankId();
5    request.addRequest(id);
6 }
7 repast::requestAgents<Package>(context, request, provider,
   receiver);
```

Listing 8: Example Agent Request

An AgentRequest is created, passing the process rank of the process making the request in the constructor. The requested ids are then added to the request. Then the request is made. Repast HPC will call the providers on the appropriate processes to provide the Packages that encapsulate the requested agents, and transfer these Packages to the receivers. The receivers will create Agents from these and Repast HPC will add those to the context.

Agent state synchronization works similarly, except that rather than creating new agents the receiver is expected to update existing agents from the Package contents.

```
template<typename Package, typename Provider, typename
    Receiver>
void repast::syncAgents(Provider& provider, Receiver& updater)
```

The RepastProcess component also synchronizes the status of an agent. If an agent has been removed from the simulation (e.g. "died") then any processes with a copy of it must be informed of its new status. Similarly, if an agent migrates to another process and becomes local to a new process then its status must be synchronized.

```
template<typename T, typename Content, typename Provider,
    typename AgentCreator>
void RepastProcess::syncAgentStatus(SharedContext<T>& context,
     Provider& provider, AgentCreator& creator)
```

Additional details about these methods and functions as well as the template type requirements can be found in the API documentation.

3.2 Parallel Scheduling

As described above, Repast HPC uses a discrete event scheduler to iterate the simulation forward. Each process has its own discrete-event schedule. The schedules are tightly and conservatively synchronized across processes (i.e., all processes execute the same tick). At each iteration of the schedule loop, each schedule determines the next tick (time step) that should be executed. Each schedule only executes events scheduled for a global minimum next tick which is determined using MPI's 'all reduce on the set of local next ticks. This is largely invisible to the user; the user is free to simply schedule events as needed; Repast HPC handles any necessary synchronization. Events are typically agent behaviors and data collection.

3.2.1 Scheduling an Event

The RepastProcess provides access to the ScheduleRunner. The ScheduleRunner API allows the programmer to schedule events to execute at:

- At a specific time (tick)

```
ScheduledEvent* scheduleEvent(double at,
    Schedule::FunctorPtr func);
```

- Starting a specific time (tick) and repeating at some specified interval

```
ScheduledEvent* scheduleEvent(double start, double
   interval, Schedule::FunctorPtr func);
```

- At the end of a simulation run

```
void scheduleEndEvent(Schedule::FunctorPtr func);
```

Additional methods schedule the tick at which to stop the simulation and to preemptively stop the simulation. See the ScheduleRunner API docs for details.

The actual item scheduled for execution is a method call on some particular object instance. This method call is wrapped by some necessary objects. For example,

```
1 ScheduleRunner& runner = RepastProcess::instance()->
2    getScheduleRunner();
3 runner.scheduleEvent(1, 1,
4    Schedule::FunctorPtr(new MethodFunctor<MyModel> (myModel,
5    &MyModel::go)));
```

Listing 9: Scheduling Example

This schedules the execution of the *go* method on an instance of a MyModel class. The go method will be executed at tick 1 and every tick thereafter (an interval of 1). (Note that typical simulations have one class that is responsible for setup and initialization of the schedule. The MyModel here is just such a class.) All scheduled methods need to be wrapped in a MethodFunctor templated with the type that contains the method. The MethodFunctor is then wrapped by a Schedule::FunctorPtr.

3.3 SharedContext

The SharedContext is a Context implementation specialized for the parallel distributed environment. Each process must contain at least one instance of a SharedContext. The SharedContext contains all the local and non-local agents currently on that process. A SharedContext is templated with the type of the agent(s) it contains. If it contains multiple types, the then template type is a shared base class and the agent type component of the AgentId can be used to determine the concrete class type. As a collection of agents, a SharedContext has a variety of methods for adding and retrieving them. For example,

```
bool addAgent(T* agent);
T* getAgent(const AgentId& id);
void getRandomAgents(const int count, std::vector<T*>& agents)
   ;
void removeAgent(const AgentId id);
void removeAgent(T* agent);
```

There also iterators for iterating over both the local and all agents as well as agents by type.

```
const_local_iterator localBegin() const;
const_local_iterator localEnd() const;
const_iterator begin() const;
const_iterator end() const;
const_bytype_iterator byTypeBegin(int typeId) const;
const_bytype_iterator byTypeEnd(int typeId) const;
```

Lastly, Projections can be associated with the SharedContext by adding them to the SharedContext. They can also be retrieved by name.

```
void addProjection(Projection<T>* projection);
Projection<T>* getProjection(const std::string& name);
```

See the API docs for Context and SharedContext for more details.

3.4 Parallel Projections

Parallel projections impose relational structure on members of contexts. In addition to the API relevant to their the structure (e.g. Network::addEdge), parallel projections provide functionality for composing a pan-process projection from smaller

local ones. Smaller local grids create a pan-process grid through the use of a buffer Smaller local networks create a pan-process network though cross-process links, and the creation of complementary links. Parallel projections are added by name to a SharedContext during their setup. Projections can be retrieved by name from the SharedContext. Any agent added to the SharedContext is automatically added to its associated projections.

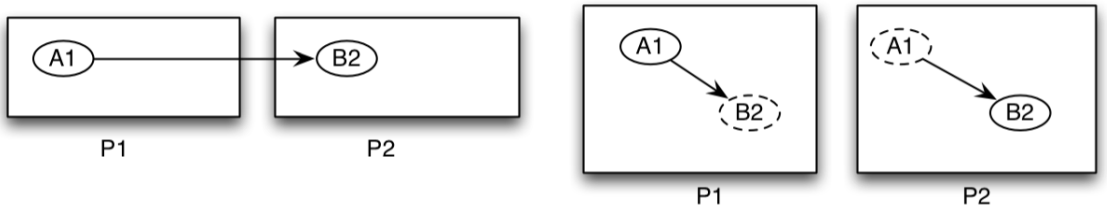
3.4.1 SharedNetwork

The SharedNetwork provides typical network functionality such as creating and removing edges, finding node successors and predecessors, iterators over vertices and edges.

```
E* addEdge(V* source, V* target);
E* addEdge(V* source, V* target, double weight);
E* findEdge(V* source, V* target);
void successors(V* vertex, std::vector<V*>& out);
void predecessors(V* vertex, std::vector<V*>& out);
void adjacent(V* vertex, std::vector<V*>& out);
```

where V and E are template parameters, identifying the vertex (i.e. agent) and edge types respectively. The default edge type is RepastEdge and any user edge types should extend the RepastEdge class.

The network represented by the SharedNetwork class is distributed across processes. Each process manages some smaller part of the larger pan-process network. The smaller parts are stitched together by the user using shared nodes(i.e. agents) and shared edges. The cross-process creation and synchronization of nodes and edges is almost completely automated by Repast HPC. For example, the simulation running on process P1 wants to create a cross-process edge between its local agent P1 and the non-local agent B2 on process P2:



In order to do so, the simulation on P1 must request B2 using the request agents method described previously. The SharedContext on P1 will then contain a non-local copy of B2 and consequently B2 will be a node in the SharedNetwork running on P1. The edge can then be created between A1 and B2 in the SharedNetwork on P1. However, in order for the complementary edge between A1 and B2 to be created in the SharedNetwork on P2, `createComplementaryEdges` must be called. This will create a copy of A1 in the SharedContext on P2 as well as add a copy of the edge to the SharedNetwork on P2.

```
template<typename Vertex, typename Edge, typename AgentContent
    , typename EdgeContent, typename EdgeManager, typename
    AgentCreator>
void createComplementaryEdges(SharedNetwork<Vertex, Edge>* net
    , SharedContext<Vertex>& context, EdgeManager& edgeManager,
     AgentCreator& creator);
```

`createComplementaryEdges` uses the Package pattern described earlier, adapted to work with edges. Sharing edges requires both the packaging of the edge as well as packaging of the agents that are the source and target nodes of the edge. For this reason `createComplementaryEdges` requires an EdgeManager that acts the provider and receiver for the edges and an AgentCreator that can receive the agent Packages that represent the edge's nodes. See the API documentation for more details.

3.4.2 Some SharedNetwork examples

Creating a network:

```
SharedNework* net = new SharedNetwork<ModelAgent, ModelEdge>(
    "network",  true);
```

The SharedNetwork constructor takes two arguments:

1. the name of the network . This should be unique with respect to the other projections in the simulation.
2. whether or not the network is directed.

The SharedNetwork also takes two template arguments:

1. the node (that is, the agent) type
2. the edge type. This must either be or extend `repast::RepastEdge`.

Once the network has been created edges can be added, network neighbors can be retrieved, etc.

```
AgentId sourceId(1, 1, 0);
AgentId targetId(3, 1, 0);
ModelAgent* source = context.getAgent(sourceId);
ModelAgent* target = context.getAgent(targetId);

SharedNetwork* net = new SharedNetwork<ModelAgent,ModelEdge>
   ( "network", true);
context.addProjection(net);
// add an edge between the source and target
net->addEdge(source, target);
// get all the agents that have an incoming edge from source
std::vector<ModelAgent*> nghs;
net->successors(source, nghs);
// do something with the successors.
```

See the SharedNetwork API documentation for more details.

3.5 Shared Grids and SharedSpaces

Shared Grid and Shared Space projections provide the typical ABMS grid and space functionality for discrete grids and continuous spaces, specifically: moving agents to some particular grid or space location; getting the agent or agents at a particular location; and getting the location of some particular agent.

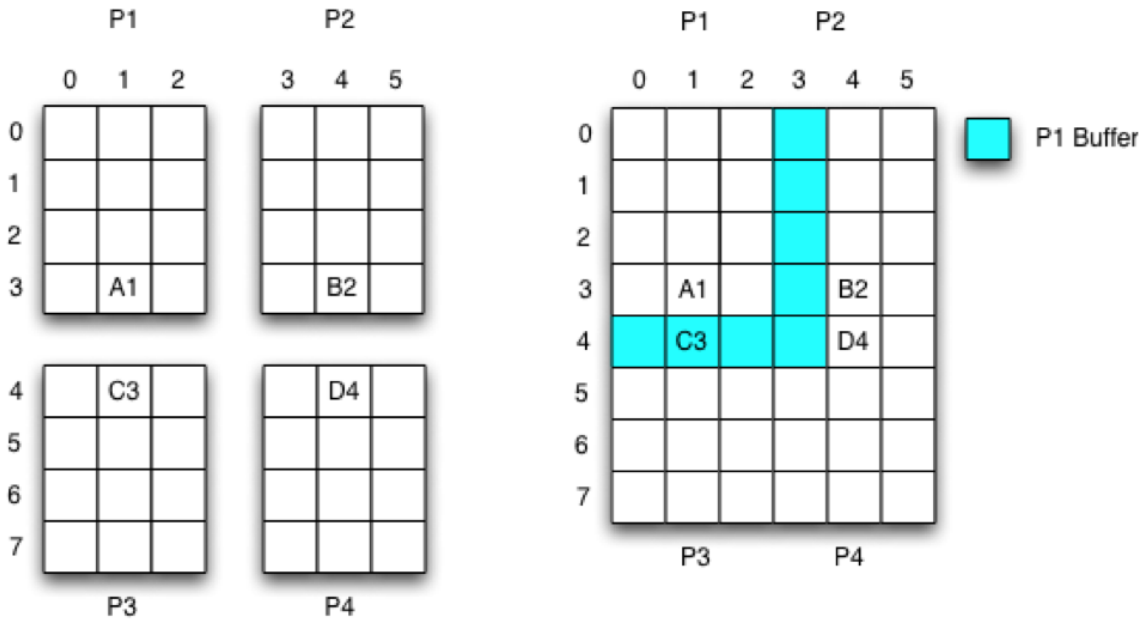
```
bool moveTo(const AgentId& id, const Point<GPType>& pt);
T* getObjectAt(const Point<GPType>& pt) const;
void getObjectsAt(const Point<GPType>& pt,
   std::vector<T*>& out) const;
bool getLocation(const T* agent,
    std::vector<GPType>& out) const;
```

`GPType` is the coordinate type for a grid or spaces location. For a discrete grid the `GPType` will be an int. For a continuous space it will be a double.

The discrete grid and continuous space code derives from a common base class and they are largely separated through the use of particular template parameters. Four typical types have been defined as typedefs with default template arguments in the SharedGrids struct:

1. SharedWrappedGrid - a discrete periodic grid, that is, a grid wrapped to become a torus.
2. SharedStrictGrid - a discrete grid with strict, non-wrapped borders.
3. SharedWrappedSpace - a continuous space wrapped into a torus.
4. SharedStrictSpace - a continuous space with strict, non-wrapped borders.

As with the SharedNetwork, shared grids and spaces are distributed across processes. Each process is responsible for a particular section of a larger pan-process grid. Each subsection has a user-defined buffer that specifies how much of the grid to share with other processes.



Here there are 4 processes, each responsible for a subsection of the grid. The entire grid runs from (0,0) x (5,7). Process P1 is responsible for the subsection (0,0) x (2, 3), P2 for (3,0) x (5,3) and so on. The buffer has a value of 1, so P1, for instance, contains a buffered area that includes all of column 3 from P2 and row 4 from P3. In this way, the larger pan-process grid is stitched together by overlapping the smaller subsections. Grid and space buffers are synchronized using the Package type pattern the `synchBuffer` method called on the grid or space that is being synchronized. See the API docs for more info.

Repast HPC is designed so that agents local to a process have a location that is within the bounds of the process's grid or space subsection. For example, all the agents local to P1 are expected to have a location within (0,0) x (2, 3). If an agent moves off the local grid, then that agent will be migrated to the process whose grid subsection the agent has moved into. This migration will occur when the agent's status is updated using `void RepastProcess::syncAgentStatus(SharedContext<T>& context, Provider& provider, AgentCreator& creator)` described above.

3.5.1 Some Grid and Space Examples

Creating a discrete grid:

```
grid = new SharedGrids<ModelAgent>::SharedWrappedGrid("grid",
   GridDimensions(Point<int>4000, 5000)), std::vector<int>(2,
   2), 2);
```

The grid (and space) constructor takes 4 arguments:

1. the name of the grid or space . This should be unique with respect to the other projections in the simulation.
2. the dimensions of the grid. This is an optional origin followed by the extent of the grid.
3. a vector indicating how many processes are in the x and y dimensions. For example, in the graphic above the X dimension has two processes and the Y also has 2. The elements of this vector must multiply to the number of processes the simulation is run with.
4. the buffer size

Using the grid:

```
1 AgentId agentId(3, 1, 0);
2 ModelAgent* agent = context.getAgent(agentId);
3 // move the agent to grid cell at 10, 10
4 grid->moveTo(agent, Point<int>(10, 10));
5
6 // get the all the agents at 10, 10 and put them in out
7 std::vector<ModelAgent*> out;
8 grid->getObjectsAt(Point<int>(10, 10), out);
```

Continuous Spaces work similarly but use `Point<double>` for locations.

```
space = new SharedGrids<ModelAgent>::SharedWrappedSpace(
   "space", GridDimensions(Point<int>4000, 5000)), std::vector
      <int>(2, 2),2);

AgentId agentId(3, 1, 0);
ModelAgent* agent = context.getAgent(agentId);
space->moveTo(agent, Point<double>(10.3, 10.3));

std::vector<ModelAgent*> out;
space->getObjectsAt(Point<double>(1324.2, 1032.3), out);
```

3.6 Data Collection

Repast HPC supports two types of data collection:

1. Aggregate Data Collection. Aggregate data collection uses MPI's reduce functionality to aggregate (sum, min, max, etc) data across processes and write the result to single file. The user specifies named numeric data sources and the reduction operation to apply to each of them across all processes. The resulting output records the tick at which the data was collected and the result of the reduce operation on each named data source. Output can be written in netCDF or plain text format. Data collection tick is written to a tick column. Each data source result is written to its own column.

2. Non-Aggregate Data Collection. Non-Aggregate data collection is similar to aggregate data collection but no reduce operation is specified. Output is to single file and includes a column variable specifying the process rank that produced the data.

Data is captured in a Dataset. The steps to creating a Dataset are:

1. Create a DataSetBuilder
2. Add Data Sources to the builder
3. Create the DataSet from the builder
4. Schedule the DataSet recording and writing.

There are different builders for the different data formats, netCDF and plain text. The process and methods are the same but the netCDF ones will be prefixed with NC while the plain text will be prefixed with SV (for Separated Value). The following is an example using the netCDF format.

```
// create a builder for netcdf aggregate data collection
NCDataSetBuilder builder("./output/data.ncf", RepastProcess::
    instance()->getScheduleRunner().schedule());
// this is the data source
InfectionSum* infectionSum = new InfectionSum(this);
builder.addDataSource(repast::createNCDataSource("
    number_infected", infectionSum, std::plus<int>()));

DataSet* dataSet = builder.createDataSet();
// schedule the record and write on the dataset
ScheduleRunner& runner = RepastProcess::instance()->
    getScheduleRunner();
runner.scheduleEvent(1.1, 1, Schedule::FunctorPtr(new
    MethodFunctor<repast::DataSet> (dataSet,
    &repast::DataSet::record)));
Schedule::FunctorPtr dsWrite = Schedule::FunctorPtr(new
    MethodFunctor<repast::DataSet> (dataSet,
    &repast::DataSet::write));
runner.scheduleEvent(100.2, 100, dsWrite);
// make sure we write the data when the sim ends
runner.scheduleAtEnd(dsWrite);
```

Here, `InfectionSum` is the data source. A data source must implement the TDataSource interface. Note that the DataSet will take care of properly disposing the InfectedSum* pointer.

```
template<typename T>
class TDataSource {
public:
virtual ~TDataSource() {};
virtual T getData() = 0;
};
```

When the data is going to be aggregated across processes then the template parameter `T` must be numeric (an int or double). See the API documentation for DataSet,

NCDataSetBuilder and SVDataSetBuilder for more details.

3.7 Random Number Generation

Repast HPC supports centralized random number generation via `Random` singleton class. This uses boot's random number library and its implementation of the MersenneTwister as a pseudo-random number generator. `Random` allows the user to create uniform, triangle, Cauchy, exponential, normal and log normal distributions. These distributions can be named and registered with `Random` and then retrieved by name for use in the simulation. See the `Random` API documentation for the details.

Named random distributions and the random seed can be set across processes either in the code or via a simple properties file format. For example,

```
1 random.seed = 1
2 distribution.uni_0_4 = double_uniform, 0, 4
3 distribution.tri = triangle, 2, 5, 10
```

The first of these properties will set the random seed to 1. The second will create a named random distribution. The distribution's name will be "uni_0_4" and it will be a double_uniform distribution from [0,4). The third line will create a triangle distribution called "tri" with a lower bound of 2, most likely value of 5, and an upper bound of 10. To create a named distribution in a properties, the property key should be `distribution.` followed by the name. The value is the type of distribution, followed by comma separated parameters. These parameters should match those in `Random` for creating distributions of that type.

Repast HPC includes a Properties class that can take properties with the above format in its constructor. The random mechanism can then be initialized using this Properties object, setting the seed and then creating distributions as described. For example,

```
Properties props("./my_properties.txt");
// initialize random from the properties file
repast::initializeRandom(props);
// get named random generator "tri" as defined in the props
   file
NumberGenerator* gen = Random::instance()->getGenerator("tri")
   ;
// do a draw from the generator
double val = gen->next();
```

4 Writing a Repast HPC C++ Model

The typical Repast HPC simulation consists of

- Some number of agent classes
- The necessary Package-type code
- A "model" class
- A `main` function

The following describes these in more details and provides examples that can be used as the basis for other models.

4.1 Agent Classes and Package code

As mentioned above, all agents must implement the Agent interface.

```
class Agent {
public:
   virtual ~Agent() {}
   virtual AgentId& getId() = 0;
   virtual const AgentId& getId() const = 0;
};
```

Here's an simple agent implementation. It has a single int variable `_state` that describes its state (though a typical agent will almost certainly have a more complicated state).

```
class ModelAgent: public repast::Agent {

private:
   repast::AgentId _id;
   int _state;

public:

   ModelAgent(repast::AgentId id, int state);
   virtual ~ModelAgent();

   int state() const {
      return _state;
   }

   void state(int val) {
      _state = val;
   }

   repast::AgentId& getId() {
      return _id;
   }

   const repast::AgentId& getId() const {
      return _id;
   }

   void flipState();
};
```

Packages are typically implemented as structs containing the minimal amount of agent state necessary to copy an agent from one process to another. The Package for our ModelAgent looks like:

```
1  struct ModelAgentPackage {
2
3     friend class boost::serialization::access;
4     template<class Archive>
5     void serialize(Archive& ar, const unsigned int version) {
6        ar & id;
7        ar & state;
8     }
9
10    repast::AgentId id;
11    int state;
12
13    repast::AgentId getId() const {
14       return id;
15    }
16 };
```

ModelAgentPackage implements the Package requirement by implementing `getId()` and by serializing the state and the id. Note that the AgentId implementation implements serialize and thus it can be serialized here without serializing its components individually.

4.2 The "Model" Class

The "Model" class doesn't implement any specific interface or extend any class. Rather, it is a convenient place to initialize and start the simulation running. A typical model class is responsible for:

- Creating a SharedContext and filling it with agents
- Creating Projections and adding them to the SharedContext
- Initializing data collection
- Scheduling the simulation actions
- Performing an initial synchronization of agents and projections

These can be done in the model classes constructor or in an init method. For example:

```
class Model {

private:

   int rank;

public:
   repast::SharedContext<ModelAgent> agents;
   repast::SharedNetwork<ModelAgent, ModelEdge>* net;
   repast::SharedGrids<ModelAgent>::SharedWrappedGrid* grid;
   repast::DataSet* dataSet;

   Model();
   virtual ~Model();
   void initSchedule();
   void step();
}
```

The constructor implementation looks like:

```
const MODEL_AGENT_TYPE = 0;
Model::Model() {
   // get the process rank of the process this Model is
      running on
   rank = RepastProcess::instance()->rank();
   // create 4 agents and add them to the context
   for (int i = 0; i < 4; i++) {
      AgentId id(i, rank, MODEL_AGENT_TYPE);
      agents.addAgent(new ModelAgent(id, rank));
   }

   // create a shared network and add it to the context
   net = new SharedNetwork<ModelAgent, ModelEdge> ("network",
      true);
   agents.addProjection(net);

   // create 40 x 60 grid, 2 process per column and 2 per row,
       with a buffer of 2
   grid = new SharedGrids<ModelAgent>::SharedWrappedGrid("grid
      ", GridDimensions(Point<int> (40, 60)),
         std::vector<int>(2, 2), 2);
   agents.addProjection(grid);

   NCDataSetBuilder builder("./output/data.ncf", RepastProcess
      ::instance()->getScheduleRunner().schedule());
   // this is the data source
   StateSum* sum = new StateSum(this);
   builder.addDataSource(repast::createNCDataSource("state_sum
      ", sum, std::plus<int>()));
   dataSet = builder.createDataSet();

   // do the initial synchronization of the grid buffer
   Provider provider(this);
   AgentsCreator creator(this);
   grid->synchBuffer<ModelPackage>(agents, provider, creator);
}
```

Note that the AgentId for each agent is created with a unique id component for each agent. The rank is passed in as is a constant representing the agent type.

The schedule is initialized in the initScheduleMethod.

```
1  void Model::initSchedule() {
2     ScheduleRunner& runner = RepastProcess::instance()->
         getScheduleRunner();
3     // stop at tick 2000, this is hardcoded here but often
         passed in via Properties
4     runner.scheduleStop(2000);
5     // call the step method on the Model every tick
6     runner.scheduleEvent(1, 1, Schedule::FunctorPtr(new
         MethodFunctor<Model> (this, &Model::step)));
7
8     // schedule the data recording and writing
9     runner.scheduleEvent(1.1, 1, Schedule::FunctorPtr(new
         MethodFunctor<DataSet> (dataSet, &DataSet::record)));
10    Schedule::FunctorPtr dsWrite = Schedule::FunctorPtr(new
         MethodFunctor<DataSet> (dataSet, &DataSet::write));
11    runner.scheduleEvent(25.2, 25, dsWrite);
12    runner.scheduleEndEvent(dsWrite);
13 }
```

Note that this schedules the model's step method. It is here that the model would iterate through agents in its context and have them do whatever is appropriate for a particular model. When iterating through agents in a context, it is crucially important that **the code only change the state of the local agents**. This can be accomplished by using the SharedContext's `local` iterators, which are ensured to iterate only over the local agents.

4.3 The `main` Function

The `main` function typically performs MPI and Repast HPC related initilization, creates the Model class and calls any necessary initialization methods on it, and tells the ScheduleRunner to run, and then calls any necessary clean up code. In addition, canonical Repast HPC executables take two arguments representing a logging config file (see the examples that come with the distribution) and a model properties file that contains properties used to initialize the Random number generator and if necessary to initialize the model. For example,

```
1  int main(int argc, char** argv) {
2     mpi::environment env(argc, argv);
3     std::string config = argv[1];
4     std::string propsfile = argv[2];
5
6     try  {
7        // initialize repast hpc with the config file
8        RepastProcess::init(config);
9        // create a Properties object from the propsfile
10       Properties props(propsfile);
11       // initialize Random
12       repast::initializeRandom(props);
13       // create and initialize the model
14       Model model();
15       model.initSchedule();
16       // Get the schedule runner and run it, starting the
            simulation proper
17       ScheduleRunner& runner = RepastProcess::instance()->
18          getScheduleRunner();
19       runner.run();
20    } catch (std::exception& ex) {
21       std::cerr << "Error while running the rumor model: " <<
            ex.what() << std::endl;
22       throw ex;
23    }
24
25    RepastProcess::instance()->done();
26    return 0;
27 }
```

The first thing that main should do is initialize the mpi environment as in line 1 above. It is also useful to wrap the code that runs the simulation in a `try` block in order to catch and display any errors that the simulation might produce while running.

5 Repast HPC and Logo-like Development

In addition to writing simulations in C++ and explicitly using the Repast HPC components, users can also write simulations in a Logo-like C++. Logo is a widely

used educational programming language commonly found in K-12 classes. Its ease of use speeds model-development, and, for many users, it is easier to conceive of and design a model using the Logo paradigm. Repast HPC Logo-like C++ further hides the complexities of implementing a parallel simulation. The Logo-like code is built upon the Repast HPC core components and all that was said there concerning agents, agent ids, parallelism and synchronization applies here as well.

5.1 Core Logo Constructs

Repast HPC uses the core Logo constructs:

- *Turtles* are the mobile agents
- *Patches* are the fixed agents
- *Links* connect turtles to form networks
- The *Observer* provides overall model management

The Logo world is a two-dimensional continuous space. Turtles can be located at any points in this space. Patches are located at the discrete integer coordinates of this space (e.g. (0,0)), one per point. Models are developed by having turtles interact with one another and with patches.

5.1.1 Turtles

Turtles are mobile agents with an AgentId, a location and a heading. Much of what turtles typically do is move using the location and heading. There are two types of turtle movement:

1. Setting the turtle's location directly. For example,

   ```
   Turtle::setxy(x,y);
   Turtle::moveTo(x, y);
   ```

2. Using a heading, move some distance along that heading. For example,

   ```
   Turtle::move(distance)
   Turtle::forward(distance)
   ```

Turtles also have additional methods for:

- Retrieving the patches or turtles at the turtle's current location
- Retrieving the patches or turtles that are the grid-wise neighbors of a turtle's patch
- Determining the neighboring patch with the maximum or minimum of some patch value and moving to that patch.

See the Turtle API documentation for the full list of a Turtle's methods.

Users create their own agents by extending the Turtle class and adding the behavior appropriate for the model.

5.1.2 Patches

Patches are immobile agents, representing a kind of "landscape" over which the Turtles travel. Patch functionality primarily consists of:

- Location type methods for retrieving the turtles on a patch and getting the grid-wise neighbors of a patch. For example,

```
template<typename PatchType>
void Patch::neighbors(AgentSet<PatchType>& out);
```

- Providing variables that represent some property of the landscape that turtles make use of. For example, if a turtle represents a sheep, the patch might implement a variable that models how much grass is available at a given sheep's location. The sheep can then eat the grass, and the value of the grass variable decreases.

If the user only needs the patch location-type functionality, the default Patch class can be used. Otherwise, the user needs to implement a new class that extends the Patch class. See the Patch API documentation for more details on Patches.

5.1.3 The Observer

The Observer functions as an abstract "model"-type class as previously discussed, implementing some basic model management. There is typically only a single Observer per process. The Observer implementation simplifies model initialization by

automatically creating the SharedContext and the Projections appropriate for a Logo simulation. It also automatically schedules `Observer::go` to execute every tick.

As part of its model management function, the Observer also contains methods for working with Turtles, creating them, retrieving them and so on. For example,

```
template<typename AgentType>
int create(size_t count);
```

`create` will create `count` number of agents of the specified type . `create` returns the type id for agents of this type. The agent type component of the AgentId of the create Turtles will match this returned id. This id can then be used to determine what agents to create when using the Package pattern. The Repast HPC makes extensive use of templates as substitute for Logos breed functionality. The type parameter is often used determine what type of agent to create, retrieve and so on.

The Observer is an abstract class that users must extend to use in a model. In particular users must override `setup` and `go`. `setup` implementations create the agents and place those agents in some model appropriate location in the world. `go` implementations will typically iterate over the agents and call the model specific behavior on them.

5.1.4 AgentSets

Repast HPC Logo models work primarily by repeatedly operating on types lists of agents known as *AgentSets*. As mentioned above, agent types are created using using object oriented inheritance from the base Turtle and Patch classes. Individual agent instances are created using Observer methods that specify the agent type to create in a C++ template. For example,

```
create<Human>(10);
```

which creates 10 agents of the Human type. AgentSets are also created using agent types in C++ templates. For example,

```
AgentSet<Human> humans;
```

When creating agents with an Observer, those agents are stored in the Observer's SharedContext. In order to work with them they must be retrieved using the Observers `get` method. When passed an AgentSet, `get` will retrieve all agents of the specified type and put them in the set. For example,

```
create<Human>(10);
AgentSet<Human> humans;
get<Human>(humans);
```

The AgentSets can be operated on using two methods `ask` and `apply`.

- `ask` executes a method on each member of the given AgentSet. For example,

  ```
  AgentSet<Human> humans;
  get(humans);
  humans.ask(&Human::step);
  ```

  which calls `step` on each Human in the humans AgentSet.

- apply executes arbitrary code on each member of the AgentSet. For example,

  ```
  AgentSet<Human> humans;
  get(humans);
  humans.apply(RandomMove(this));
  ```

  where RandomMove is a functor (i.e. it implements `operator()`) that takes a Turtle pointer and moves it to a random location.

Typical HPC Logo simulations work by retrieving AgentSets and then either calling specific methods on the elements of that set via `ask` or applying arbitrary code to those elements via `apply`.

5.2 Repast HPC and Synchronization

Repast HPC distributes the Logo world (the continuous space and patch grid) across processes. As a Repast HPC Logo model runs, cross-process synchronization is necessary under a number of conditions:

- When a turtle moves into the space controlled by another process, the agent must be moved into the other process and out of the current process
- Patches and turtles must be buffered between neighboring processes
- If cross-process links have been created, then the network must be synchronized too

Repast HPC automates most of this communication, but as with a non-Logo simulation the user must still provide Package pattern code to extract the Turtle or Patch

state and package it for transfer and to unpack the transferred package and create the appropriate Turtle or Patch from it. Note that Repast HPC expects Patches and Turtles (default or user defined) to be transferred in the same Package structure, thus care must be taken to identify the type of the agent in the package. The agent type component of the AgentId can be used for this. The following is example using three agent types: Humans, Zombies, and the default Patch type. (See the zombie example model for the full code.)

When the Zombies and Humans are created in our Observers setup method, we store the agent types.

```
1 void ZombieObserver::setup(Properties props) {
2    Observer::setup(props);
3    int count = strToInt(props.getProperty(HUMAN_COUNT_PROP));
4    humanType = create<Human> (count);
5
6    count = strToInt(props.getProperty(ZOMBIE_COUNT_PROP));
7    zombieType = create<Zombie> (count);
8    ...
9 }
```

The types are stored in humanType and zombieType. Then when unpacking an AgentPackage we query the agent type of the package and compare that against the type variables we set above. Depending on the result, we create a Zombie, Human or Patch.

```
void ZombieObserver::createAgents(std::vector<AgentPackage>&
    contents,
    std::vector<RelogoAgent*>& out) {
    for (size_t i = 0, n = contents.size(); i < n; ++i) {
        AgentPackage content = contents[i];
        if (content.type == zombieType) {
            out.push_back(new Zombie(content.getId(), this));
        } else if (content.type == humanType) {
            out.push_back(new Human(content.getId(), this,
                content));
        } else {
            // its a patch.
            out.push_back(new Patch(content.getId(), this));
        }
    }
}
```

The AgentPackage struct:

```
struct AgentPackage {
    template<class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & id;
        ar & proc;
        ar & type;

        ar & infectionTime;
        ar & infected;
    }

    int id, proc, type;

    int infectionTime;
    bool infected;

    repast::AgentId getId() const {
        return repast::AgentId(id, proc, type);
    }
};
```

In this Package we store the id components separately and `type` refers to the agent

type.

5.3 Writing a Repast HPC Logo Model

Writing a Repast HPC Logo style Model consists of the following steps:

- Extending Turtle to implement your model specified agent or agents. (In some cases, using the default Turtle class may be enough.)
- Extend Patch if necessary
- Extend Observer, implementing `setup` and `go`.
- Create a simulation run using the SimulationRunner and run it.

To illustrate these steps, we are going to work with the Zombie example model included in the source. The Zombie model has two types of agents: Humans and Zombies. Zombies are attracted to Humans and attempt to infect them. Humans run away from Zombies. If infected, a Human becomes a Zombie after some period of time.

5.4 Turtles

Our Turtles are the Zombies and Humans. The Zombie and Human code is relatively simple.

```
1  class Zombie : public repast::relogo::Turtle {
2  public:
3     Zombie(repast::AgentId id, repast::relogo::Observer* obs) :
4     repast::relogo::Turtle(id, obs) {}
5
6     virtual ~Zombie() {}
7     // zombie behavior executed every iteration
8     void step();
9     void infect(Human* human);
10 };
```

```
class Human : public repast::relogo::Turtle {

private:
   bool _infected;
   int _infectionTime;

public:
   Human(repast::AgentId id, repast::relogo::Observer* obs):
   repast::relogo::Turtle(id, obs), _infected(false),
      _infectionTime(0) {}
   Human(repast::AgentId id, repast::relogo::Observer* obs,
   const AgentPackage& package): repast::relogo::Turtle(id,
      obs), _infected(package.infected),
         _infectionTime(package.infectionTime) {}

   virtual ~Human() {}

   // human behavior executed every iteration.
   void step();
   void infect();
};
```

Both Zombie and Human inherit from Turtle. The Human has an additional constructor that allows us to initialize it using an AgentPackage.

The Zombie implementation:

```
struct CountHumansOnPatch {
   double operator()(const Patch* patch) const {
      AgentSet<Human> set;
      patch->turtlesOn(set);
      return set.size();
   }
};

void Zombie::step() {
   // get the neighbors of the patch I'm on
   AgentSet<Patch> nghs = patchHere<Patch>()->neighbors<Patch
      >();
   // which patch as the most humans
   Patch* winningPatch = nghs.maxOneOf(CountHumansOnPatch());
   // face that patch and move towards it
   face(winningPatch);
   move(.5);

   // get the humans on my patch
   AgentSet<Human> humans;
   turtlesHere(humans);

   // if there are any get the first one and infect it
   if (humans.size() > 0) {
      Human* human = humans.oneOf();
      infect(human);
   }
}

void Zombie::infect(Human* human) {
   human->infect();
}
```

The method call details can be looked up in the Turtle and Observer API documentation. The general idea is that the Zombie is on a Patch and gets the neighboring patches in an AgentSet. It queries that AgentSet for the Patch that returns the maximum value when `CountHumansOnPatch` is applied to it. It then faces and moves towards that Patch. If there are any Humans on the Patch the Zombie is now on, then infect one of them.

The Human implementation:

```
struct CountZombiesOnPatch {
   double operator()(const Patch* patch) const {
      AgentSet<Zombie> set;
      patch->turtlesOn(set);
      return set.size();
   }
};

void Human::infect() {
   _infected = true;
}

void Human::step() {
   bool alive = true;
   if (_infected) {
      // infected, increase the amount of time infected
      _infectionTime++;
      if (_infectionTime == 5) {
         // infected for 5 tiimesteps, so hatch a zombie here
         // and kill the human.
         _observer->hatch<Zombie> (this);
         die();
         alive = false;
      }
   }

   if (alive) {
      // are there any zombies in the neighborhood
      AgentSet<Patch> nghs = patchHere<Patch> ()->neighbors<
         Patch> ();
      //  find the path with the least zombies
      Patch* winningPatch = nghs.minOneOf(CountZombiesOnPatch
         ());
      // face the patch with the least zombies
      face(winningPatch);
      // run there.
      move(1.5);
   }
}
```

As with the Zombie code, the method call details can be looked up in the Turtle and Observer API documentation. The basic idea is simple: if the Human is infected, we increase the number of time steps it has been infected and, if it has been infected long enough, turn it into a Zombie; if it remains alive (that is, not yet a zombie) then we find the neighboring Patch with the least Zombies and move there.

Note that this code has been complicated by the parallel nature of the program. It would perhaps be more natural to move the Human and then do the infection check. This would eliminate the need for `alive` bool and the conditional using it. However, if in its movement, a Human moves outside of the local bounds of the space subsection managed by its process then it will be marked for migration to that process (as described in the grid / space component section above). If subsequent to that movement though, the Human has died, it cannot then be migrated. There are other ways around this problem. For example, the Zombie and Human could be the same agent type and so there would be no reason to call `die` on the Human and remove it from the simulation. However, this code is also intended to illustrate the use of `die` and so the above solution is reasonable.

5.5 Patch

The default Patch type is appropriate for this model, so the basic Patch class is not extended.

5.6 Observer Implementation

As mentioned above, an Observer implementation must implement `go` and override `setup`. The ZombieObserver inherits from Observer and implements these methods as follows.

```
1  void ZombieObserver::setup(Properties props) {
2     Observer::setup(props);
3     int count = strToInt(props.getProperty(HUMAN_COUNT_PROP));
4     humanType = create<Human> (count);
5
6     count = strToInt(props.getProperty(ZOMBIE_COUNT_PROP));
7     zombieType = create<Zombie> (count);
8
9     AgentSet<Human> humans;
10    get(humans);
11    humans.apply(RandomMove(this));
12
13    AgentSet<Zombie> zombies;
14    get(zombies);
15    zombies.apply(RandomMove(this));
16
17    SVDataSetBuilder svbuilder("./output/data.csv", ",", repast
         ::RepastProcess::instance()->getScheduleRunner().
         schedule());
18    InfectionSum* iSum = new InfectionSum(this);
19    svbuilder.addDataSource(repast::createSVDataSource("
         number_infected", iSum, std::plus<int>()));
20    // add the create dataset to this observer.
21    addDataSet(svbuilder.createDataSet());
22 }
```

Note that a Properties object is passed to `setup` automatically. This Properites object is created from a properties file passed to the executable and will be described in more detail below. All `setup` implementations must begin by passing this Properties object to the superclass's `setup`; this allows the base Observer to perform some required initialization. The remaining code uses property values to determine how many zombies and humans to create, and then moves them to random locations. The final section of code creates a plain text dataset and adds it to the ZombieObserver. The DataSet must be created as described in the preceeding DataSet section. However, Observer automatically takes care of the scheduling once the DataSet has been added to it.

```
1  void ZombieObserver::go() {
2     // get the zombies and call step on them
3     AgentSet<Zombie> zombies;
4     get(zombies);
5     zombies.ask(&Zombie::step);
6
7     // get the humans and call step on them
8     AgentSet<Human> humans;
9     get(humans);
10    humans.ask(&Human::step);
11
12    // perform cross process synchronization
13    initSynchronize();
14    synchronizeTurtleStatus<AgentPackage> (*this, *this);
15    synchronizeTurtleCrossPMovement();
16    synchronizeBuffers<AgentPackage> (*this, *this);
17
18    if (_rank == 0) {
19       std::cout << RepastProcess::instance()->
            getScheduleRunner().currentTick()  << std::endl;
20    }
21 }
```

As mentioned above `go` is scheduled to execute every tick of the simulation, and so the above code executes every iteration. It gets AgentSets of Zombies and Humans and calls the `step` method on each of them. It closes with necessary cross-process synchronization code. This begins with a required initialization call, then synchronizes the agent status (whether they have moved or died), then it migrates any moved agents, then it synchronizes the buffers between the process grid subsections. Lastly, we print out the current tick count.

5.7 Creating a Simulation Run and Writing `main`

The SimulationRunner provides a way to automatically initialize a Repast HPC Logo simulation given a properties file, and then to run the simulation. The properties files must have the following properties defined:

- min.x the minimum integer x coordinate of the world
- min.y the minimum integer y coordinate of the world

- max.x the maximum integer x coordinate of the world
- max.h the maximum integer y coordinate of the world
- grid.buffer the size of the grid and space buffers
- proc.per.x the number of processes to assign to the world's x dimension. proc.per.x multiplied by proc.per.y must equal the number processes that the simulation will run on
- proc.per.y the number of processes to assign to the world's y dimension. proc.per.x multiplied by proc.per.y must equal the number processes that the simulation will run on
- stop.at the tick at which to stop the simulation

The SimulationRunner will use this properties to create a Logo world of the specified size, distribute the world in the specified processes configuration, and schedule the simulation to stop at the specified tick. The SimulationRunner has a single method `run` to which the Properties object should be passed.

```
template<typename ObserverType, typename PatchType>
void run(Properties& props);
```

The template parameters refer to the type of Observer to create and the type of Patches to use. Calling `run` will

- create the specified Observer
- initialize Random using the Properties instance
- call `setup` on that Observer passing the Properties instance
- create the world
- populate the world with Patches of the specified type
- schedule simulation stop
- schedule `go` to execute every tick
- run the schedule and thus start the simulation

A SimulationRunner should be created and used in a `main` function. The Zombie model `main` looks like:

```
1
2  void runZombies(std::string propsFile) {
3     Properties props(propsFile);
4     SimulationRunner runner;
5     runner.run<ZombieObserver, Patch>(props);
6  }
7
8  int main(int argc, char **argv) {
9     mpi::environment env(argc, argv);
10    std::string config = argv[1];
11    std::string props = argv[2];
12
13    RepastProcess::init(config);
14    runZombies(props);
15
16    RepastProcess::instance()->done();
17    return 0;
18 }
```

Note that some error checking code has been removed for clarity. As with a Repast HPC straight C++ model, main needs to begin by initializing the mpi environment and passing a config file to RepastProcess to initialize it. This then calls runZombies and uses a SimulationRunner to run the simulation.

6 Synchronization

Synchronization is potentially the most confusing part of a Repast HPC simulation, whether written in straight C++ or in Logo-like C++. If any there is any sharing across processes then some sort of synchronization must take place.

If agents are copied across processes via an AgentRequest.

- Agent State must be synchronized (unless you are sure the state has not changed).

  C++: `repast::syncAgents(Provider, Updater);`
  Logo: `void Observer::synchronizeTurtleState(Provider, Updater);`

- Agent Status must be synchronized (unless you are sure that no agent has died).

C++: `repast::syncAgentStatus(SharedContext, Provider, AgentCreator);`
Logo: `void Observer::synchronizeTurtleStatus(Provider, AgentCreator);`

If the simulation contains a discrete grid or continuous space projection (which all Logo-like simulations do):

- Agent Status must be synchronized (unless you are sure that no agent has moved into another process).

  C++: `repast::syncAgentStatus(SharedContext, Provider, AgentCreator);`
  Logo: `void Observer::synchronizeTurtleStatus(Provider, AgentCreator);`

- Movement must be synchronized in order to synchronize agent migration between processes due to grid or space movement.

  C++: `synchMove()` called on each grid or space that requires synchronization.
  Logo: `Observer::synchronizeTurtleCrossPMovement();`

If the simulation contains a discrete grid or continuous space projection (which, again, all Logo-like simulations do) and the buffer size is greater than 0, then the following need to be called in order:

- Initialize the buffer synchronization

  C++: `initSynchBuffer()` on each grid or space that requires buffer synchronization.
  Logo: `Observer::initSynchronize();`

- Synchronize the buffers

  C++: `synchBuffer(SharedContext, Provider, AgentsCreator)` on each grid or space that requires buffer synchronization.
  Logo: `Observer::synchronizeBuffers(Provider, AgentsCreator);`

Edges need to be synchronized if cross-process edges have been created using `createComplimentaryEdges`.

- If the state of the edge has been updated,

  C++: `synchEdges(SharedNetwork, EdgeManager);`

- If the edge (link) has has been removed:

  C++: `SharedNetwork::synchRemovedEdges()`
  Logo: `Observer::synchronizeRemovedLinks(std::string)`

Some additional caveats:

- Synchronizing an agent's status should come before synchronizing its state.
- Synchronizing agent's state and status should occur before synchronizing cross process movement.
- An agent should move only once between calls to synchronize status in order to correctly manage the effects of cross-process migration.

While Repast HPC does simplify much of the parallel programming of an agent simulation, users should still give adequate thought to how the simulation can be properly implemented to run in the parallel environment.